Google technical interview guide
Filter for prospective research software engineering roles
Spring 2022 – March 25, 2022 (tentative)

Interview tips and review material assembled from assorted sources

Last compiled with LaTeX2e on February 14, 2022

Contents

1 Interview procedure tips and tricks **2**

2 Programmatic topics in Python3 **5**

3 Data structures and sorting algorithms cheat sheets **7**

4 Sorting algorithms – Core material reference **8**

5 Data structures – Core material reference (hash tables) **11**

6 Data structures – Core material reference (linked lists, stacks, queues and heaps) **18**

7 Data structures – Core material reference (trees and traversals) **19**

8 Data structures – Core material reference (graphs) **20**

9 Recursion, backtracking and memoization – Core material reference **21**

10 Other algorithmic type preparation **23**

11 NP-completeness and standard problems **25**

12 Operating systems and systems programming mechanics **26**

13 OOP fundamentals **28**

14 Functional programming constructs **29**

15 Architecture and compilers **30**

16 Languages and formal grammars **31**

17 Basic discrete math and statistics **32**

---

1 Interview procedure tips and tricks

1.1 Six steps guide

Now that you know how prepared you need to be, figure out where you are right now. Use CTCI for this. Take a couple of questions from each section and solve them using the six steps I mentioned earlier. Keep track of how long it takes you to reach an optimal solution for each problem you solve.

1.1.1 1 – Repeat the question in your own words

As soon as you hear it, repeat it out loud. Do this preferably in your own words to demonstrate your comprehension. Remember, your interviewer is there to assist you, so repeating the question aloud will only serve to make their job (and yours) easier.

This will also buy you some time to think about what you've been asked and develop good questions or approaches. Plus, hearing yourself restate the problem might help you think through it more clearly.

Clarify the problem statement and explain clearly the working approach. They are looking for thought process.

1.1.2 2 – Check assumptions

Many candidates start writing code almost immediately after hearing the question. This is a big mistake. Most coding questions have some level of ambiguity built in. Have at least two or three questions ready so that you can confirm you have all the necessary detail to solve the problem. You can also write out confirmed assumptions on the whiteboard (do it in small print to save space). Questions like "*Does input fit in memory?*" or "*Can I assume input is always valid?*" are good candidates if you can't think of any at first.

Other examples of good clarifying questions to ask:

- Are there any time or space complexity requirements?
- How is the input data set specified? In an array? If so, is the array sorted or otherwise structured?
- What is the data set type and range? Integers, floating points numbers? Does it include positive, negative and zero valued inputs? Is the valid range bounded?
- What data needs to be returned? True/False for success/existence, or the actual data witnessing the property is true?

1.1.3 3 – Use real examples

You've got a whiteboarduse it! Draw out an example array, a binary tree, a linked list, etc. Give it real data and write out the expected output of a working solution. You should practice coming up with good examples as part of your study regime (you do have one, right?). Using a visual example also gives you opportunity to think up more questions and your interviewer a chance to correct your assumptions. Don't forget to keep thinking out loud as you work through it.

If your interviewer provides examples, use those since they probably exist for a reason. This is also how interviewers will point you towards problems with your design or implementation.

1.1.4 4 – Brainstorm solutions and their time/space complexity

Stop and think about various approaches. Think about trade offs using Big-O analysis and think out loud. Don't stop with the first solution that comes to mind. Always ask yourself what's the best you can do. Trust me, we always will!

Tips: Don't forget the spacetime tradeoff principle. Wanna go fast? Use more space. Also, hash tables people! And learn how to think graphsincluding trees. If all else fails, recursion, backtracking, and memoization are useful tools on particularly difficult problems.

1.1.5 5 – Write working code (no pseudo-code please!)

You might normally use pseudo-code to design your code, but you don't have time for that in a 45-minute interview. Choose your strongest language and turn your thoughts into working code as quickly as possible. This should be the easiest part of the interview assuming you've put time into practicing without an IDE (seriously, don't use an IDE to practice for the interview unless you're compiling code you've already written out). Make sure to practice using a whiteboard or just use pen and paper.

1.1.6 6 – Test your code, always

Now turn your brain into a compiler and execute each line of code to ensure you don't have any logical bugs. Use the whiteboard to keep track of variable state as you iterate. Also, use the example(s) you created earlier and confirm that you get the expected output. Testing is crucial in software engineering, so never make the grave mistake of leaving this step out.

1.1.7 7* – Benchmark yourself (days 3–5)

Now that you know how prepared you need to be, figure out where you are right now. Use CTCI for this. Take a couple of questions from each section and solve them using the six steps I mentioned earlier. Keep track of how long it takes you to reach an optimal solution for each problem you solve.

If and only if youve solved the problem yourself, take a look at the accompanying solution to assess how you did. Did you reach the optimal solution or at least progress beyond the naive/brute force answer? How long did it take you? Was your code written in the fewest lines possible?

Do this for every section. When done, you can prioritize the sections that you didnt do so well on up front in your practice regimen and leave the other sections for later. You should repeat this exercise just before your interview so that you know your weak spots going into the day of the interview.

1.2 Tips and graph theoretic thinking and modeling

Whenever someone gives you a problem, *think graphs*. They are the most fundamental and flexible way of representing any kind of a relationship, so it's about a 50-50 shot that any interesting design problem has a graph involved in it. Make absolutely sure you can't think of a way to solve it using graphs before moving on to other solution types. **This tip is important!**

1.3 Videos of feedback and live interviews

Start with the sourcecheck out `google.com/careers` for info on how Google hires. Then watch this video **(1)** from Google about what interviewers look for in the interview, and finally check out an example interview **(2)** featuring real Google engineers.

(1) `https://www.youtube.com/watch?v=ko-KkSmp-Lk`

(2) `https://www.youtube.com/watch?v=XKu_SEDAykw`

TODO What no one tells you about coding interviews (why leetcode doesn't work)

TODO My Google Interview — Offer accepted

1.4 Other links and postings online

- How should I prepare for my Google software interview if I have one month left?
- Interviewing at Google? Here's 6 Things You Absolutely Need To Do
- My Guide To Preparing for the Google Technical Interview
- Coding practice: `www.leetcode.com`
- Coding practice: `www.interviewcake.com`

1.5 Have a friend interview you

The friend should ask you a random interview question, and you should go write it on the board. You should keep going until it is complete, no matter how tired or lazy you feel. Do this as much as you can possibly tolerate.

Practice a few times with another person, both with someone technical and someone non-technical. Ask them if:

- You looked and sounded relaxed;
- You looked like a disciplined problem solver;
- You kept thinking out loud throughout the entire exercise?

Sleep and DO NOT be arrogant: *You should go in humble, open-minded, and focused.*

1.6 Environment questions for the virtual interview

- Can I share my screen with a tablet writing device?
- Can I refer to printed notes I have prepared for the interview?

2 Programmatic topics in Python3

2.1 Getter and setter instance methods

2.2 Math library functionality

2.3 Built-in list types

Some common operations on a list include:

```
## INITIALIZATION:
lst = []
lst = [ constValue ] * lstLength
lst = strLiteral.split('')

## COMMON OPERATIONS:
lst.insert(index, value)
lst.remove(value)
lst.pop(index)
lst.find(value) # < 0 if NOT found, else gives the first index

## SORTING (default algorithm) is in-place (the list itself is modified) and
## stable (the initial ordering of two equal elements is maintained):
lst.sort(key=lambda elt: func(elt), reverse=True|False)
newList = sorted(lst, key=lambda xarr: (x[0], x[1]), reverse=True|False)
```

2.4 The collections library functionality

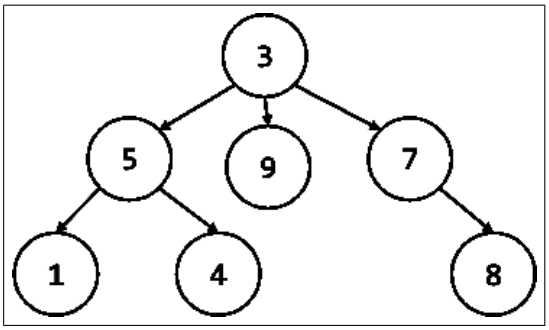
2.5 Implementation of the default dictionary structures

2.6 Tree implementation with nested dictionaries

Because Python doesn't implement trees for us, we'll define them using *dictionaries*. Each node of the tree will be a dictionary that has two keys:

- The first key is the string "`value`", which maps to the value in the node.
- The second key is the string "`children`", which maps to a list of dictionaries, the child nodes. Leaves have empty child lists.

Consider the following binary tree and its associated dictionary-based tree structure:




```
t = { "value" : 3, "children" :
     [
        { "value" : 5, "children" :
           [
              { "value" : 1, "children" : [ ] },
              { "value" : 4, "children" : [ ] }
           ]
        },
        { "value" : 9, "children" : [ ] },
        { "value" : 7, "children" :
           [
              { "value" : 8, "children" : [ ] }
           ]
        }
     ]
}
```

2.7 Priority queues (PQ)

2.8 Unordered sets

2.9 Multisets

2.10 Functional programming

TODO: Know implementation and tradeoffs for use;
Running times for the internal methods ???

2.11 Testing

Unit tests; design tests for corner or interesting cases of the input data;

3 Data structures and sorting algorithms cheat sheets

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Common Data Structure Operations

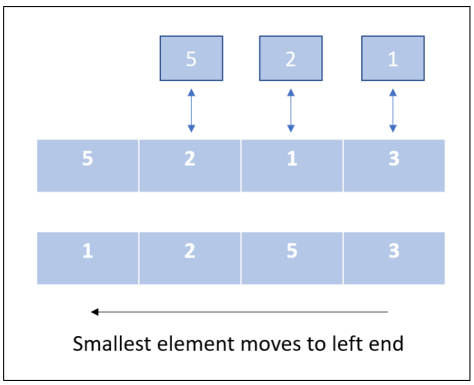
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	θ(1)	θ(n)	θ(n)	θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	θ(1)	θ(1)	θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

4 Sorting algorithms – Core material reference

4.1 Selection sort

Summary: NON-RECURSIVE & UNSTABLE & IN-PLACE & WC: $O(n^2)$

In this algorithm, we create two segments of the list one sorted and the other unsorted. We continuously remove the smallest element from the unsorted segment of the list and append it to the sorted segment. We dont swap intermediate elements. Hence this algorithm sorts the array in the minimum number of swaps.

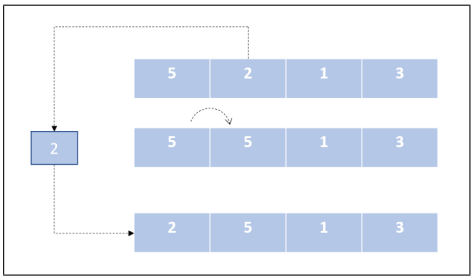



```
def selectionSort(array):
    for i in range(len(array)-1):
        min_idx = i
        for idx in range(i + 1, len(array)):
            if array[idx] < array[min_idx]:
                min_idx = idx
        array[i], array[min_idx] = array[min_idx], array[i]
    return array
```

4.2 Insertion sort

Summary: NON-RECURSIVE & STABLE & IN-PLACE & WC: $O(n^2)$

Like Selection Sort, in this algorithm, we segment the list into sorted and unsorted parts. Then we iterate over the unsorted segment and insert the element from this segment into the correct position in the sorted list.

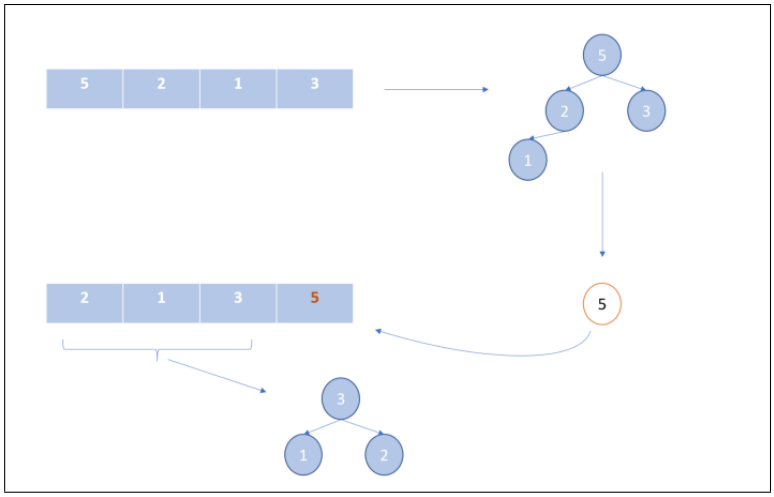



```
def insertionSort(array):
    for i in range(1, len(array)):
        key = array[i]
        j = i-1
        while array[j] > key and j >= 0:
            array[j+1] = array[j]
            j -= 1
        array[j+1] = key
    return array
```

4.3 Heap sort

Summary: NON-RECURSIVE & UNSTABLE & IN-PLACE & WC: $O(n \log n)$

We create two segments of the list one sorted and one unsorted. In this, we use the heap data structure to efficiently get the max element from the unsorted segment of the list. The heapify method uses recursion to get the max element at the top.

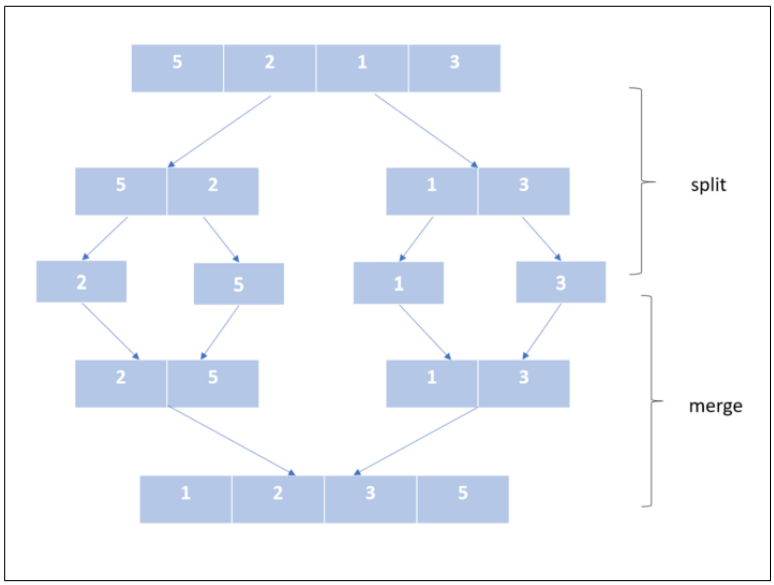



```
def heapify(array, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and array[i] < array[l]:
        largest = l
    if r < n and array[largest] < array[r]:
        largest = r
    if largest != i:
        array[i], array[largest] = array[largest], array[i]
        heapify(array, n, largest)
def heapSort(array):
    n = len(array)
    for i in range(n//2, -1, -1):
        heapify(array, n, i)
    for i in range(n-1, 0, -1):
        array[i], array[0] = array[0], array[i]
        heapify(array, i, 0)
    return array
```

4.4 Merge sort

Summary: RECURSIVE & STABLE & NEEDS-EXTRA-SPACE & WC: $O(n \log n)$

This is a divide and conquer algorithm. In this algorithm, we split a list in half and keep splitting the list by 2 until it only has a single element. Then we merge the sorted list. We keep doing this until we get a sorted list with all the elements of the unsorted input list.

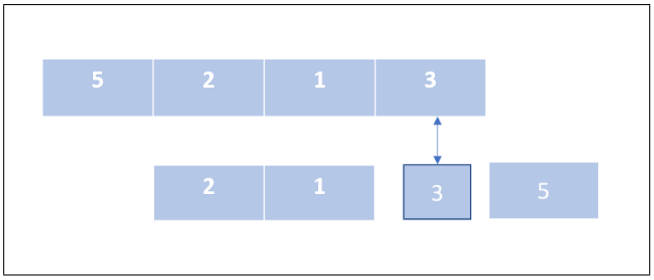



```
def mergeSort(nums):
    if len(nums)==1:
        return nums
    mid = (len(nums)-1) // 2
    lst1 = mergeSort(nums[:mid+1])
    lst2 = mergeSort(nums[mid+1:])
    result = merge(lst1, lst2)
    return result
def merge(lst1, lst2):
    lst = []
    i = 0
    j = 0
    while(i<=len(lst1)-1 and j<=len(lst2)-1):
        if lst1[i]<lst2[j]:
            lst.append(lst1[i])
            i+=1
        else:
            lst.append(lst2[j])
            j+=1
    if i>len(lst1)-1:
        while(j<=len(lst2)-1):
            lst.append(lst2[j])
            j+=1
    else:
        while(i<=len(lst1)-1):
            lst.append(lst1[i])
            i+=1
    return lst
```

4.5 Quick sort

Summary: RECURSIVE & UNSTABLE & IN-PLACE & WC: $O(n \log n)$

In this algorithm, we partition the list around a pivot element, sorting values around the pivot. Here, the last element from the list as a pivot value. Best performance is achieved when the pivot value splits the list into two almost equal halves.




```
def quickSort(array):
    if len(array)> 1:
        pivot=array.pop()
        grtr_lst, equal_lst, smlr_lst = [], [pivot], []
        for item in array:
            if item == pivot:
                equal_lst.append(item)
            elif item > pivot:
                grtr_lst.append(item)
            else:
                smlr_lst.append(item)
        return (quickSort(smlr_lst) + equal_lst + quickSort(grtr_lst))
    else:
        return array
```

4.5.1 Selection of the pivot elements

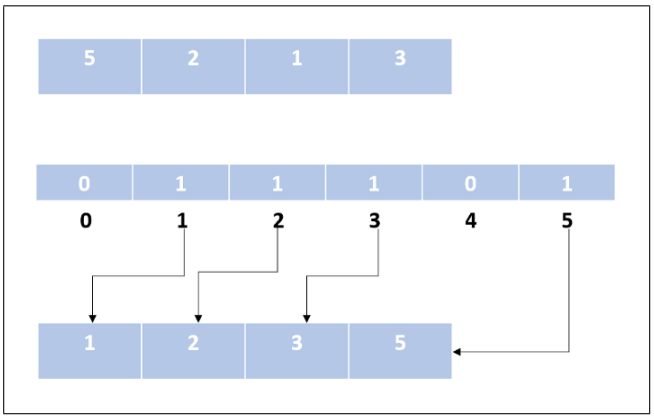
As you can see, Quicksorts efficiency often depends on the pivot selection. If the input array is unsorted, then using the first or last element as the pivot will work the same as a random element. But if the input array is sorted or almost sorted, using the first or last element as the pivot could lead to a worst-case scenario. Selecting the pivot at random makes it more likely Quicksort will select a value closer to the median and finish faster.

Another option for selecting the pivot is to find the median value of the array and force the algorithm to use it as the pivot. This can be done in $O(n)$ time. Although the process is little bit more involved, using the median value as the pivot for Quicksort guarantees you will have the best-case Big–O scenario.

4.6 Counting sort (on non-negative integers)

Summary: NON-RECURSIVE & STABLE & IN-PLACE-WITH-EXTRA-SPACE & WC: $O(n)$

This algorithm does not do a comparison between the elements. We use properties of the integers to sort. We count how many times a number has come and store the count in the array where the index is mapped to the keys value.




```
def countingSortArray(self, nums: List[int]) -> List[int]:
    i_lower_bound , upper_bound = min(nums), max(nums)
    lower_bound = i_lower_bound
    if i_lower_bound < 0:
        lb = abs(i_lower_bound)
        nums = [item + lb for item in nums]
        lower_bound , upper_bound = min(nums), max(nums)

    counter_nums = [0]*(upper_bound-lower_bound+1)
    for item in nums:
        counter_nums[item-lower_bound] += 1
    pos = 0
    for idx, item in enumerate(counter_nums):
        num = idx + lower_bound
        for i in range(item):
            nums[pos] = num
            pos += 1
    if i_lower_bound < 0:
        lb = abs(i_lower_bound)
        nums = [item - lb for item in nums]
    return nums
```

5 Data structures – Core material reference (hash tables)

5.1 Hash tables

A *hash table* organizes data so you can quickly look up values for a given key.

Pros:

- **Fast lookups:** Lookups take $O(1)$ time on average. In many cases, hash tables have average case operation running times that are more efficient than search trees or other table-based lookup structures (e.g., arrays).
- **Operations:** Insert and deletion are $O(1)$ on average, but $O(n)$ in the worst cases. Space requirement is $O(n)$.
- **Flexible key types:** Most data types can be used for keys, as long as they're hashable.

Cons:

- **Slow (linear-time) worst case lookups:** Lookups take $O(n)$ time in the worst case.
- **Unordered:** Keys aren't stored in a special order. If you're looking for the smallest key, the largest key, or all the keys in a range, you'll need to look through every key to find it.
- **Single-directional lookups only:** While you can look up the value for a given key in $O(1)$ time, looking up the keys for a given value requires looping through the whole dataset – $O(n)$ time.
- **Not cache-friendly:** Many hash table implementations use linked lists, which don't put data next to each other in memory.

5.1.1 Lookup, insert and removal operations

The advantage of using hashing is that the table address of a record can be directly computed from the key. Hashing implies a function h, when applied to a key k, produces a hash M. However, since M could be potentially large, the hash result should be mapped to finite entries in the hash table – or slots – several methods can be used to map the keys into the size of hash table N. The most common method is the division method, in which modular arithmetic is used in computing the slot.

$$h(k) = M \bmod N.$$

This is often done in two steps. Namely, compute

1. `Hash` = `HashFunction`(`Key`); then
2. `Index` = `Hash` (mod `HashTableSize`).

The running times of these functions are appealing in the *average case*, which can be determined by *amortized analysis*, or an average cost of a single operations run m times.

- Suppose that the hash function h is uniformly distributed so that given any key k, $\mathbb{P}(h(k) \bmod N \equiv i) = \frac{1}{N}$ for $0 \leqslant i < N$ the array indexed that k is initially hashed into.
- Let the indicator random variable χ_i be defined such that for k in the key space $\chi_i(k) = [h(k) \bmod N \equiv i]_\delta$.
- Suppose that we want the expected running time of m lookups of the keys $\{k_i\}_{1 \leqslant i \leqslant m}$. We compute that

$$\begin{aligned} \mathbb{E}[\texttt{Lookup}(k_m)] &= \frac{1}{m} \times \sum_{0 \leqslant i < N} \mathbb{E}[\chi_i(k_1) + \cdots + \chi_i(k_m)] = \frac{1}{m} \times \sum_{0 \leqslant i < N} \sum_{1 \leqslant j \leqslant m} \mathbb{E}[\chi_i(k_j)] \\ (*) \qquad &= \frac{1}{m} \times \sum_{\substack{0 \leqslant i < N \\ 1 \leqslant j \leqslant m}} \mathbb{P}(h(k_j) = i) = \frac{n}{m \cdot N}. \end{aligned}$$

- Then under the assumption that we can resize the array structure of size N to another capacity at least as large as $2N$ while keeping the uniformity property of the hash function in amortized constant time, we can keep the last ratio in $(*)$ bounded within any fixed range we like. That is, the lookup operation can be made $O(1)$ on average.
- Small modifications show that the average running times of the insert and removal operations on the hash table run in $O(1)$ expected time, with a worst possible case of $O(n)$ – linear in the number of keys inserted so far.

5.1.2 Choosing the array size

A basic requirement is that the *hash function* should provide a (nearly) uniform distribution of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g., a Pearson's chi-squared test for discrete uniform distributions.

The distribution needs to be uniform only for table sizes that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of the table size, then the hash function needs to be uniform only when the size is a power of two. Here the index can be computed as some range of bits of the hash function. On the other hand, some hashing algorithms prefer to have the size be a prime number. The modulus operation may provide some additional mixing; this is especially useful with a poor hash function.

5.1.3 Hash table load and rehashing – Design statistics

The *load factor* of a hash table is defined to be

$$\alpha \equiv \texttt{HashTableLoadFactor}(n,k) := \frac{n}{k},$$

where n is the number of array entries occupied in the hash table and k is the number of available buckets. The perfromance of lookups suffers as $\alpha \to 1$. It is also inefficient to resize the hash table when the load factor is too small to justify the operation. A practical ideal value of α might be between 0.6 and 0.75.

Repeated insertions cause the number of entries in a hash table to grow, which consequently increases the load factor; to maintain the amortized $O(1)$ performance of the lookup and insertion operations, a hash table is dynamically resized and the items of the tables are rehashed into the buckets of the new hash table,[13] since the items cannot be copied over as varying table sizes results in different hash value due to modulo operation. Resizing may be performed on hash tables with fewer entries compared to its size to avoid excessive memory usage.

If the design requires that the array size be prime. We need to find the next prime number that is greater than $2N+1$ where N is the current (prime-valued) size of the array. Finding the next such prime can be performed in $O\left(N^{\frac{3}{2}}\right)$ time (see this link, in section *finding the next prime*, for example).

5.1.4 Collision resolution: Open addressing and separate chaining

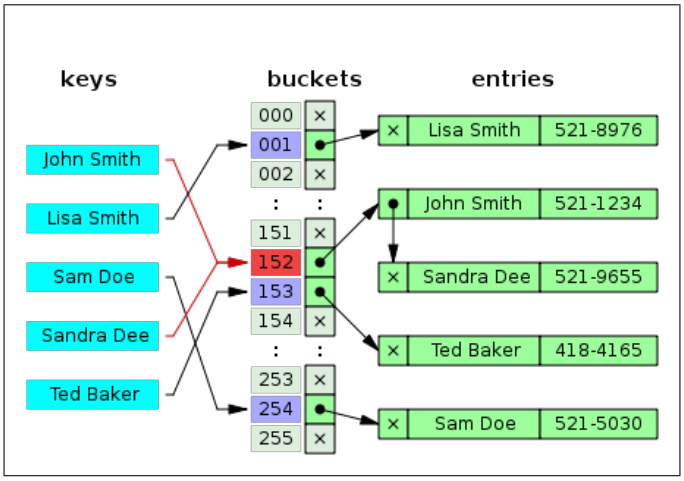
This strategy is also called *two-choice hashing*. The first part is to compute the hash function on a key which resolves the initial array index of that key. The second step is to perform *collision resolution* when distinct keys still collide at the same array index. Common collision resolution strategies include *open hashing* (or *open addressing*) and *separate chaining* (or just *chaining*).

Discussion of separate chaining: The basic API for operations is

```
ChainedHashInsert(HT, x):
    insert x at the head of the linked list HT[h(x.key)]
ChainedHashSearch(HT, k):
    search for element with key k in linked list HT[h(k)]
ChainedHashDelete(HT, x):
    delete x from the linked list located at HT[h(x.key)]
```

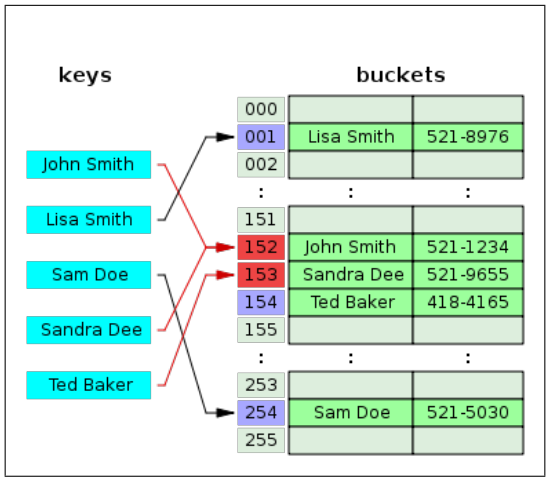
If the keys of the elements are ordered, it's efficient to insert the item by maintaining the order when the key is comparable either numerically or lexically, thus resulting in faster insertions and unsuccessful searches. However, the

standard method of using a linked list is not cache-conscious since there is little spatial localitylocality of referencesince the nodes of the list are scattered across the memory, hence doesn't make efficient use of CPU cache. In cache-conscious variants, a dynamic array found to be more cache-friendly is used in the place where a linked list or self-balancing binary search trees is usually deployed for collision resolution through separate chaining, since the contiguous allocation patten of the array could be exploited by hardware-cache prefetcherssuch as translation lookaside buffer-resulting in reduced access time and memory consumption.



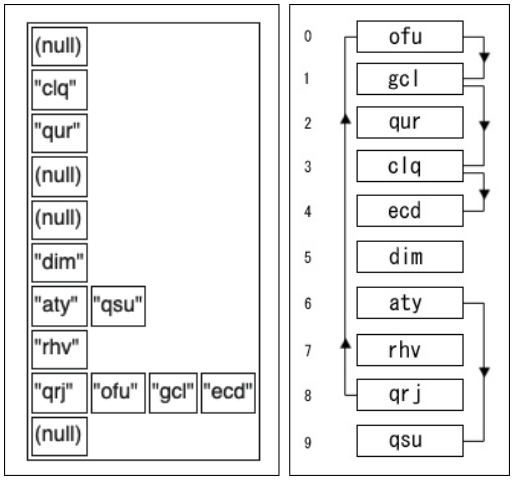
Discussion of open addressing:

All entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Well-known probe sequences include: linear probing, quadratic (polynomial) probing and double hashing where a second hash function is used to resolve the next available slot in the array.



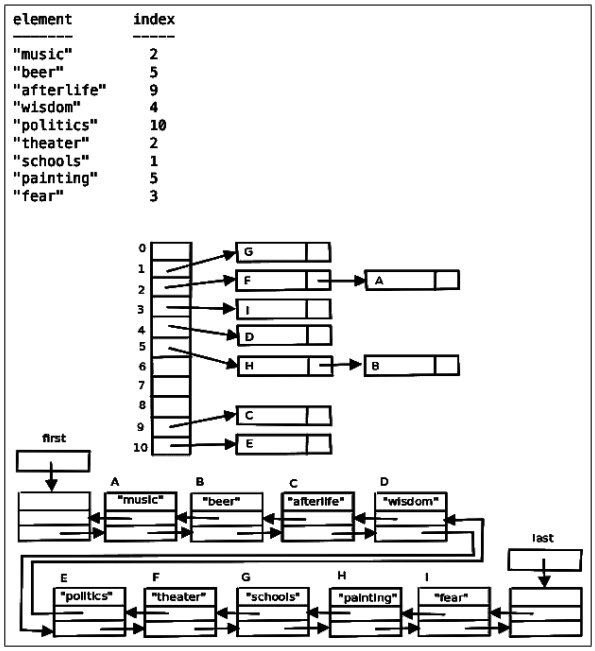
Discussion of coalesced hashing:

Coalesced hashing is a hybrid of both separate chaining and open addressing in which the buckets or nodes link within the table. The algorithm is ideally suited for fixed memory allocation. This method is similar to separate chaining in that it gives a pointer parameter to each bucket that tracks the location of the next collision node, creating a traceable chain of each node that collided at the original index.



5.1.5 Linked hash sets

The purpose of the *linked hash set* object, as we have seen, is to keep track of the entry order of elements while retaining its $O(1)$ average times for the lookup-based operations. The way to achieve this effect is to maintain, in addition to the hash table, a (doubly) linked list to hold the actual elements. Hence,we can use this structure variant to not only get the benefits of a hash table, but also provide an iterator to the keys preserving the original order (or something else) in which the keys have been inserted into the table.



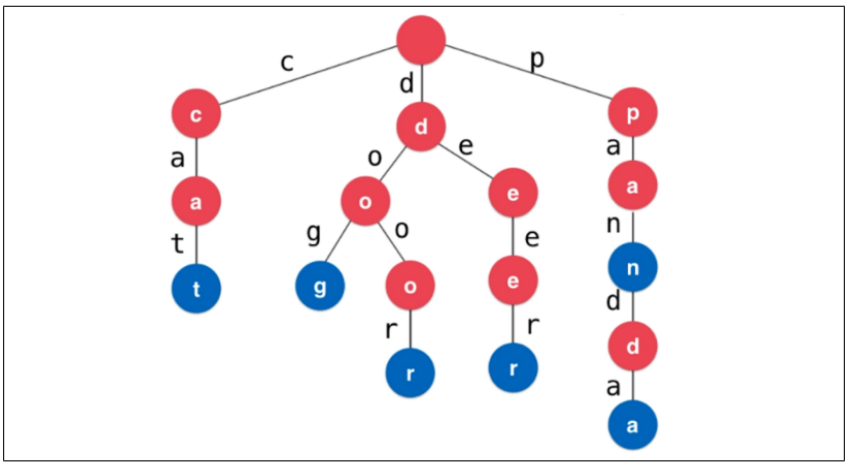
5.2 Hash sets

A *hash set* is an implementation of the *unordered set* interface which does not allow duplicate value.

5.3 Hash maps

A *hash map* is an implementation of the *map* interface which maps a key to a value (lookups stored via a *hash table* mechanism). Note that duplicate keys are forbidden in a *map* instance.

5.4 Dictionary trees – Trie-trees – or digital trees – or prefix trees




```
Structure Node
    Children Node[AlphabetSize]
    IsTerminal Boolean
    Value DataType
```

```
## WORST-CASE:   O(m) where m is thee length of the key to search for
## AVERAGE-CASE: O(log_m(n)) where n is the total number of keys and m
##               is the alphabet size
TrieFind(x, key)
    for i in range(0, len(key) + 1):
        if x.Children[key[i]] == None:
            return False
        x = x.Children[key[i]]
    return x.Value
## WORST-CASE (TIME):  O(|w| * n) where w is the search key and n is the
##                     number of keys in the tree
## WORST-CASE (SPACE): O(|w| + n) [...]
TrieInsert(x, key, value)
    for i in range(0, len(key) + 1):
        if x.Children[key[i]] == None:
            x.Children[key[i]] = Node()
        x = x.Children[key[i]]
    x.Value := value
    x.IsTerminal = True
TrieDelete(x, key)
    if key == None:
        if x.IsTerminal:
            x.IsTerminal = False
            x.Value = None
        return None
    x.Children[key[0]] = TrieDelete(x.Children[key[0]], key[1:])
    return x
```

5.4.1 Trie-trees as a replacement for hash tables

- Searching for a node with an associated key of size m has the complexity of $O(m)$, whereas an imperfect hash function may have numerous colliding keys, and the worst-case lookup speed of such a table would be $O(N)$, where N denotes the total number of nodes within the table.
- Tries do not need a hash function for the operation, unlike a hash table; there are also no collisions of different keys in a trie.
- Buckets in a trie, which are analogous to hash table buckets that store key collisions, are necessary only if a single key is associated with more than one value.

- String keys within the trie can be sorted using a predetermined alphabetical ordering.

However, tries are less efficient than a hash table when the data is directly accessed on a secondary storage device such as a hard disk drive that has higher random access time than the main memory. Tries are also disadvantageous when the key value cannot be easily represented as string, such as floating point numbers.

5.4.2 Compressed versions – see radix trees below

A *radix tree*, also known as a *compressed trie*, is a space-optimized variant of a trie in which nodes with only one child get merged with its parents; elimination of branches of the nodes with a single child results in better in both space and time metrics. This often works best under conditions where the trie remains static and set of keys stored are very sparse within their representation space.

5.4.3 External memory tries

Several trie variants are suitable for maintaining sets of strings in external memory, including suffix trees. A combination of trie and B-tree, called the B-trie has also been suggested for this task; compared to suffix trees, they are limited in the supported operations but also more compact while performing update operations faster.

5.4.4 Applications

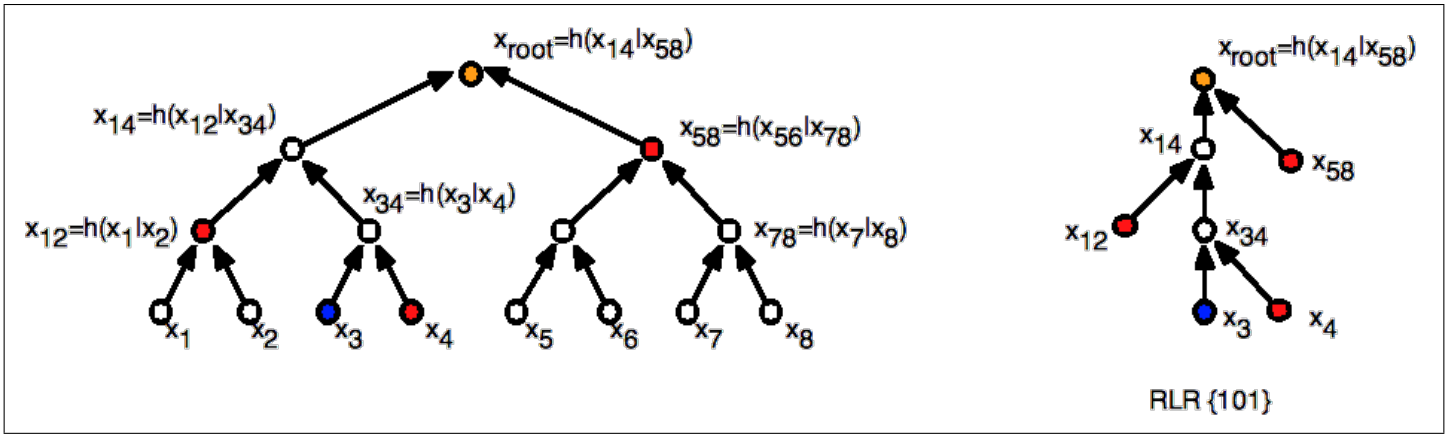
- **Lexicographical sorting:** Lexicographic sorting of a set of string keys can be implemented by building a trie for the given keys and traversing the tree in pre-order fashion; this is also a form of radix sort. Tries are also fundamental data structures for burstsort, which is notable for being the fastest string sorting algorithm as of 2007, which is accompanied for its efficient use of CPU cache.
- **Web search engines:** A specialized kind of trie called a compressed trie, is used in web search engines for storing the indexes – a collection of all searchable words. Each terminal node is associated with a list of URLs – called occurrence list – to pages that match the keyword. The trie is stored in the main memory, whereas the occurrence is kept in an external storage, frequently in large clusters.
- **Internet routing:** Compressed variants of tries, such as databases for managing Forwarding Information Base (FIB), are used in storing IP address prefixes within routers and bridges for prefix-based lookup to resolve mask-based operations in IP routing.

5.5 Hash chains

Most *cryptographic hash functions* are designed to take a string of any length as input and produce a fixed-length hash value. The security level of a cryptographic hash function has been defined using the following properties:

- **Pre-image resistance:** Given a hash value h, it should be difficult to find any message m such that $h = \text{hash}(m)$. This concept is related to that of a one-way function.
- **Second pre-image resistance:** Given an input m_1, it should be difficult to find a different input m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$. This property is sometimes referred to as weak collision resistance.
- **Collision resistance:** It should be difficult to find two different messages m_1 and m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$. Such a pair is called a cryptographic hash collision. This property is sometimes referred to as strong collision resistance. It requires a hash value at least twice as long as that required for pre-image resistance; otherwise collisions may be found by a *birthday attack*.

A *hash chain* is a successive application of a cryptographic hash function h to a string x. For example, $h(h(h(x))) \equiv h^3(x)$ is a hash chain of length 3. Binary hash chains are commonly used in association with a *hash tree*. A binary hash chain takes two hash values as inputs, concatenates them and applies a hash function to the result, thereby producing a third hash value (cf. Merkle trees).



The RLR (101) *order bits* are necessary to complete the hash chain above.

5.6 Unordered sets

A *(unordered) set* is like a hash map except it only stores keys, without values. Usually, we're interested in whether something is in a set or not. Sets are usually implemented very similarly to hash maps – using hashing to index into an array – but they don't have to worry about storing values alongside keys. These are implemented using hash maps – each member of the set is a key in the hash map with a dummy value that gets ignored.

5.7 Bloom filters

TODO: See `https://www.interviewcake.com/concept/python/bloom-filter?`

6 Data structures – Core material reference (linked lists, stacks, queues and heaps)

6.1 Doubly versus singly linked lists – Usage and operation running times

6.2 Dynamic arrays (and expected constant time operations with resizing)

6.3 Skip lists

6.4 Stacks

6.5 Queues (generic container)

6.5.1 Binomial queues

Other types of common – See Java API ???

TODO: See `https://en.wikipedia.org/wiki/Priority_queue`

6.6 Heaps

TODO: Min-heaps ???

6.6.1 Priority queues

7 Data structures – Core material reference (trees and traversals)

7.1 Binary and n-ary trees

Examples of usage:

- Directory structures in a filesystem;
- Company organizational heirarchy;
- Sports tournament brackets.

7.2 Tree traversals

7.2.1 BFS and DFS

7.2.2 In-order, pre-order and post-order traversals

7.2.3 Sorting (generic trees)

The length of the maximal path through the tree on n nodes is $n+1$. A generic sorting algorithm on any tree (without knowing more about its underlying structure) is therefore $O(n)$ in the worst case.

7.3 Binary search trees (BSTs)

TODO: See CMU site slides ¿= page 21;

7.4 Balanced (binary) trees

7.4.1 Red-Black trees

7.4.2 Splay trees

7.4.3 AVL trees

7.4.4 B-trees ???

TODO: See `https://en.wikipedia.org/wiki/B-tree`

7.5 Radix trees (Patricia tries)

TODO: See `https://en.wikipedia.org/wiki/Radix_tree`

7.6 k-D trees

7.7 Quad trees

8 Data structures – Core material reference (graphs)

8.1 Implementations and representations in code

8.1.1 Objects and pointers

8.1.2 Matrix-based storage

8.1.3 Adjacency lists

8.2 BFS and DFS

8.3 Dijkstra's algorithm (TODO – Purpose)

TODO: See `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`
TODO: For running time charts by graph type case, see `https://en.wikipedia.org/wiki/Shortest_path_problem#Single-source_shortest_paths`

8.4 Prim's algorithm (TODO – Purpose)

TODO: See `https://en.wikipedia.org/wiki/Prim%27s_algorithm`

8.5 Kruskal's algorithm (TODO – Purpose)

TODO: See `https://en.wikipedia.org/wiki/Kruskal%27s_algorithm`

8.6 Bellman-Ford algorithm (TODO – Purpose)

TODO: See `https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm`

8.7 Floyd-Warshall algorithm (all pairs shortest paths)

8.8 A^* algorithm

TODO: See `https://en.wikipedia.org/wiki/A*_search_algorithm`

8.9 General purpose algorithms

8.9.1 Distance

8.9.2 Search problems

8.9.3 Graph connectivity problems

8.9.4 Cycle-detection algorithms

8.10 Topological sort

8.11 Probabilistic methods and expected running times and optimal solution guarantees

9 Recursion, backtracking and memoization – Core material reference

9.1 Dynamic programming – Top-down versus bottom-up approaches

Top-down approach: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.

Bottom-up approach: Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of F_{41} and F_{40}, we can directly calculate the value of F_{42}.

9.2 Backtracking – Includes general procedure pseudocode

Backtracking is a general algorithm for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("*backtracks*") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

9.2.1 Description of the method

The backtracking algorithm enumerates a set of *partial candidates* that, in principle, could be *completed* in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of *candidate extension* steps.

Conceptually, the partial candidates are represented as the nodes of a tree structure, the *potential search* tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.

The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the whole sub-tree rooted at c is skipped (*pruned*). Otherwise, the algorithm (1) checks whether c itself is a valid solution, and if so reports it to the user; and (2) recursively enumerates all sub-trees of c. The two tests and the children of each node are defined by user-given procedures.

Therefore, the *actual search tree* that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is the number of nodes of the actual tree times the cost of obtaining and processing each node. This fact should be considered when choosing the potential search tree and implementing the pruning test.

9.2.2 Pseudocode implementation (generic algorithm)

In order to apply backtracking to a specific class of problems, one must provide the data P for the particular instance of the problem that is to be solved, and six procedural parameters. These procedures should take the instance data P as a parameter and should do the following:

1. `root(P)`: return the partial candidate at the root of the search tree.
2. `reject(P, c)`: return true only if the partial candidate c is not worth completing.
3. `accept(P, c)`: return true if c is a solution of P, and false otherwise.
4. `first(P, c)`: generate the first extension of candidate c.
5. `next(P, s)`: generate the next alternative extension of a candidate, after the extension s.
6. `output(P, c)`: use the solution c of P, as appropriate to the application.

The backtracking algorithm reduces the problem to the call `backtrack(root(`P`))`, where `backtrack` is the following recursive procedure:

```
procedure backtrack(c) is
    if reject(P, c) then return
    if accept(P, c) then output(P, c)
    s <- first(P, c)
    while s != NULL do
        backtrack(s)
        s <- next(P, s)
```

Early stopping variants: The pseudo-code above will call output for all candidates that are a solution to the given instance P. The algorithm can be modified to stop after finding the first solution, or a specified number of solutions; or after testing a specified number of partial candidates, or after spending a given amount of CPU time.

9.2.3 Examples

- Constraint satisfaction of a boolean-valued function on m integer variables;
- See blog article: Backtracking, Memoization & Dynamic Programming!

9.3 Memoization

9.4 Dynamic programming

9.5 Greedy algorithms

TODO: See `https://en.wikipedia.org/wiki/Greedy_algorithm#Submodular_functions`

10 Other algorithmic type preparation

10.1 Divide and conquer

10.2 Prime factorizations

10.3 Huffmann coding

10.4 Error correcting codes

10.5 Hamming codes

10.6 Variable-length codes

TODO: See `https://en.wikipedia.org/wiki/Variable-length_code`

10.7 Best known algorithms for matrix operations

10.8 Matrix decompositions – SVD, LU, QR, Cholesky (LDL), Householder, etc.

10.9 Least squares, linear least squares and best fit modeling problems

Let X be a $n \times p$ matrix, β be a $p \times 1$ column vector and y a $n \times 1$ column vector. In general, this is an over-determined system of n linear equations in p unknown coefficients (the rows of β) satisfying

$$\sum_{1 \leqslant j \leqslant p} X_{ij}\beta_j = y_i, \text{ for } i = 1, 2, \ldots, n.$$

This system can be expressed in matrix form as $X\beta = y$. Since such a system usually cannot be solved exactly, we seek to find the coefficients $\{\beta_i\}_{1 \leqslant i \leqslant p}$ that are the "*best fit*", here with respect to the following quadratic minimization problem:

$$\hat{\beta} = \operatorname{argmin}_\beta S(\beta), \text{ where } S(\beta) := \sum_{1 \leqslant i \leqslant n} \left| y_i - \sum_{1 \leqslant j \leqslant p} X_{ij}\beta_j \right|^2 \equiv ||y - X\beta||^2.$$

This particular form of the minimization problem has a unique solution whenever the p columns of X are linearly independent:

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T y = \beta + \left(X^T X\right)^{-1} X^T \varepsilon.$$

10.10 Sequence alignment – Dynamic programming application

The problem can be stated naturally as a recursion, a sequence A is optimally edited into a sequence B by either:

1. Inserting the first character of B, and performing an optimal alignment of A and the tail of B;
2. Deleting the first character of A, and performing the optimal alignment of the tail of A and B;
3. Replacing the first character of A with the first character of B, and performing optimal alignments of the tails of A and B.

The partial alignments can be tabulated in a matrix, where cell (i, j) contains the cost of the optimal alignment of $A[1..i]$ to $B[1..j]$. The cost in cell (i, j) can be calculated by adding the cost of the relevant operations to the cost of its neighboring cells, and selecting the optimum.

10.11 Locality of reference

Locality of reference, also known as the *principle of locality*, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time. There are two basic types of reference locality – *temporal* and *spatial locality*. Temporal locality refers to the reuse of specific data and/or resources within a relatively small time duration. Spatial locality (also termed *data locality*) refers to the use of data elements within relatively close storage

locations. *Sequential locality*, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as traversing the elements in a one-dimensional array.

11 NP-completeness and standard problems

11.1 Precise definition of the NP-complete problem class

11.2 Traveling salesman problem (TSP)

The *travelling salesman problem* (also called the *travelling salesperson problem* or *TSP*) asks the following question: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*
It is an NP-hard problem in combinatorial optimization.

TODO: See `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

11.3 Knapsack problem

TODO: See `https://en.wikipedia.org/wiki/Knapsack_problem`

11.4 Subset sum problem (SSP)

TODO: See `https://en.wikipedia.org/wiki/Subset_sum_problem`

11.5 Partition problem

The *partition problem*, or *number partitioning*, is the task of deciding whether a given multiset S of positive integers can be partitioned into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2. This problem corresponds to the SSP where all inputs are positive, and the target sum is exactly half of all inputs: $T = \frac{1}{2}(a_1 + \cdots + a_n)$.

11.6 Independent sets (graph theory)

TODO: See `https://en.wikipedia.org/wiki/Independent_set_(graph_theory)`

11.7 Set cover problem (SCP)

TODO: See `https://en.wikipedia.org/wiki/Set_cover_problem`

11.8 Bin packing problem (BPP)

TODO: See `https://en.wikipedia.org/wiki/Bin_packing_problem`

11.9 Multiway number partitioning

TODO: See `https://en.wikipedia.org/wiki/Multiway_number_partitioning`

11.10 Randomized algorithmic approaches

11.10.1 The Monte-Carlo method

TODO: See `https://en.wikipedia.org/wiki/Monte_Carlo_method`

11.10.2 Graph coloring type problems ???

12 Operating systems and systems programming mechanics

12.1 Systems design (SWE specs)

Questions are used to assess a candidate's ability to combine knowledge, theory, experience and judgement toward solving a real-world engineering problem. In other words, can a candidate "design policies, processes, procedures, methods, tests, and/or components from ground up.

Sample topics include: Features sets, interfaces, class hierarchies, distributed systems, designing a system under certain constraints, simplicity, limitations, robustness and tradeoffs. You should also have an understanding of how the internet actually works and be familiar with the various pieces (routers, domain name servers, load balancers,firewalls, etc.). Other topics include: traverse a graphs, run-time complexity of graphs, distributed hash table system, resource estimation with real systems, big product design picture, translation of an abstract problem to a system, API discussions, binary trees, cache, mapreduce, for loop problems, index, reverse link-list, compilers, memory cache, networks and also common topics.

12.2 Operating systems (SWE specs and advice from online)

SWE statement: You should understand processes, threads, concurrency issues, locks, mutexes, semaphores, monitors and how they all work. Understand deadlock, livelock and how to avoid them. Know what resources a process needs and a thread needs (also: resourcem allocation for processes and threads). Understand how context switching works, how it's initiated by the operating system and underlying hardware. Know a little about scheduling. We are rapidly moving towards multi-core, so know the fundamentals of "modern" concurrency constructs.

BlogSpot tips: This is just a plug, from me, for you to know about processes, threads and concurrency issues. A lot of interviewers ask about that stuff, and it's pretty fundamental, so you should know it. Know about locks and mutexes and semaphores and monitors and how they work. Know about deadlock and livelock and how to avoid them. Know what resources a processes needs, and a thread needs, and how context switching works, and how it's initiated by the operating system and underlying hardware. Know a little about scheduling. The world is rapidly moving towards multi-core, and you'll be a dinosaur in a real hurry if you don't understand the fundamentals of "modern" (which is to say, "kinda broken") concurrency constructs.
Source: `http://steve-yegge.blogspot.com/2008/03/get-that-job-at-google.html`

12.3 Types of design tradeoffs – Examples

12.3.1 Lookup tables vs. recalculation

A common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

12.3.2 Compressed vs. uncompressed data

A spacetime tradeoff can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical. There are also rare instances where it is possible to directly work with compressed data, such as in the case of compressed bitmap indices, where it is faster to work with compression than without compression.

12.3.3 Re-rendering vs. stored images

Storing only the SVG source of a vector image and rendering it as a bitmap image every time the page is requested would be trading time for space; more time used, but less space. Rendering the image when the page is changed and storing the rendered images would be trading space for time; more space used, but less time. This technique is more generally known as caching.

12.3.4 Smaller code vs. loop unrolling

Larger code size can be traded for higher program speed when applying loop unrolling. This technique makes the code longer for each iteration of a loop, but saves the computation time required for jumping back to the beginning of the loop at the end of each iteration.

12.4 Systems – The basics of systems programming

12.5 Scheduling algorithms

12.6 Internet and networking basics

12.6.1 Routers

12.6.2 DNS and hierarchy

12.6.3 Load balancers

12.6.4 Firewalls

12.7 Distributed systems

12.8 Flow networks (graphs)

TODO: See `https://en.wikipedia.org/wiki/Flow_network`

13 OOP fundamentals

13.1 Abstraction

13.2 Inheritance

13.3 Cohesion

13.4 Coupling

14 Functional programming constructs

14.1 Map-Reduce

14.2 Reap-Sow

15 Architecture and compilers

15.1 Compiling and linking

15.2 Cache and memory cache

TODO: See `https://en.wikipedia.org/wiki/CPU_cache`

15.3 LRU cache

TODO: See `https://www.interviewcake.com/concept/python/lru-cache?`

16 Languages and formal grammars

16.1 Grammars

16.2 Lexers and tokenizers

16.3 Parsers and syntax flow rules

17 Basic discrete math and statistics

17.1 Counting – Four types chart

17.2 Probability review

17.2.1 Independence

Two events A, B are *independent* if $\mathbb{P}(A \cap B) = \mathbb{P}(A) \times \mathbb{P}(B)$. A collection of events $\{A_i\}$ are *mutually, or pairwise, independent* provided that A_i and A_j are independent whenever $i \neq j$.

17.2.2 Conditional probability and Bayes theorem

The *conditional probability* of the event A given the event B is defined to be

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)},$$

where $\mathbb{P}(A|B) = \mathbb{P}(A)$ whenever A, B are independent.

Suppose that we have a collection of events $\{A_i\}$. Then *Baye's theorem* states that (cf. posterior probabilities)

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}, \text{ if } \mathbb{P}(B) \neq 0.$$

Often this can be combined with the next identity when B is the union of the pairwise disjoint events $\{B_i\}_{i \geqslant 1}$:

$$\mathbb{P}(B) = \sum_{i \geqslant 1} \mathbb{P}(A|B_i)\mathbb{P}(B_i).$$

17.2.3 Conditional distributions and conditional expectation

We have that $\mathbb{E}[Y|Y] = Y$ and $\mathbb{E}\mathbb{E}[X|Y] = \mathbb{E}[X]$. Precisely, the *conditional distribution of X given Y* is defined to be

(Discrete RVs)
$$\mathbb{E}[X|Y = y] = \sum_x \frac{\mathbb{P}(X = x \wedge Y = y)}{\mathbb{P}(Y = y)},$$

(Continuous RVs)
$$= \frac{1}{f_Y(y)} \times \int_{-\infty}^{\infty} x f_{X,Y}(x, y) dx,$$

where f_Y is the marginal distribution of Y and $f_{X,Y}$ is the joint distribution of X and Y. We also have that for a continuous random variable X with PDF f_X,

$$\mathbb{P}(A) = \int_{-\infty}^{\infty} \mathbb{P}(A|X = x) f_X(x) dx.$$

17.2.4 Markov's inequality and formulas for first moments

If $X \geqslant 0$, then

$$\mathbb{E}[X] = \int_0^{\infty} \mathbb{P}(X \geqslant \lambda) d\lambda.$$

A generalized form of *Markov's inequality* states that for any $a > 0$ and any Borel function f

$$\mathbb{P}(X \geqslant a) \leqslant \frac{\mathbb{E}[f(X)]}{f(a)}.$$

The *first moment method* is based on the property that if $X \geqslant 0$, then $\mathbb{E}[X] = 0 \iff X = 0$ almost surely.

17.2.5 Chebyshev's inequality and the second moment method

We have the following variants of *Chebyshev's inequality* for $\mathrm{Var}(X) \equiv \sigma^2 < +\infty$ where $\mu := \mathbb{E}[X]$ and the parameters $k, \epsilon > 0$:

- $\mathbb{P}(|x - \mu| \geqslant k\sigma) \leqslant \frac{1}{k^2}$.
- $\mathbb{P}(|x - \mu| \geqslant \epsilon) \leqslant \frac{\sigma^2}{\epsilon^2}$.
- $\mathbb{P}(|x - \mu| < \epsilon) > 1 - \frac{\sigma^2}{\epsilon^2}$.

The *second moment method* is based on the property that if $X \geqslant 0$ and X has finite variance, then

$$\mathbb{P}(X > 0) \geqslant \frac{\mathbb{E}[X]^2}{\mathbb{E}[X^2]}.$$

17.2.6 Laws of large numbers (weak and strong)

The WLLN can be applied in some cases, i.e., when Lebesgue integration predicts no first moment exists for a random variable, where the SLLN cannot be applied. The SLLN states that for $\bar{X}_n := \frac{1}{n}(X_1 + \cdots + X_n)$ with $\{X_i\}_{i \geqslant 1}$ a sequence of IID random variables that are integrable so that their expected value exists and is finite, we have

$$\bar{X}_n \xrightarrow{a.s.} \mathbb{E}[X_1].$$

The variances $\mathrm{Var}(X_i)$ need not be finite. The WLLN states that for any fixed $\epsilon > 0$,

$$\lim_{n \to \infty} \mathbb{P}\left(|\bar{X}_n - \mathbb{E}[X_1]| < \epsilon\right) = 1.$$

Note that the WLLN allows that $|\bar{X}_n - \mathbb{E}[X_1]| \geqslant \epsilon$ infinitely often, though at increasingly infrequent intervals as $n \to \infty$. In contrast, when the SLLN applies, for any fixed $\epsilon > 0$ at all sufficiently large $n \geqslant N(\epsilon)$, we must have that $|\bar{X}_n - \mathbb{E}[X_1]| < \epsilon$.

17.2.7 Central limit theorem

Let $\{X_i\}_{i \geqslant 1}$ be a sequence of IID random variables with first moment μ and variance $\sigma^2 < +\infty$. We have the convergence in distribution to

$$\lim_{n \to \infty} \sqrt{n}\left(\frac{\bar{X}_n - \mu}{\sigma}\right) \sim \mathcal{N}(\imath, \infty).$$

17.2.8 Borel-Cantelli lemma – TODO

17.2.9 Caratheodory criterion

$$\mu(B) = \mu(B \cap A) + \mu(B \backslash A)$$

17.2.10 Monotone convergence theorem for expectation

17.3 Basic statistical calculations

17.3.1 Types of averages and inequalities

We have the following types of means for a collection $\{a_i\}_{1 \leqslant i \leqslant n}$ of n numbers:

- **Arithmetic mean:** TODO
- **Geometric mean:** TODO
- **Harmonic mean:** TODO
- **Root mean square (RMS) average:** TODO

The *AGM inequality* states that TODO ...

17.3.2 Variance and covariance properties

The *variance* of a random variable X is defined to be $\mathrm{Var}(X) := \mathbb{E}[X^2] - \mathbb{E}[X]^2$. We have that $\mathrm{Var}(X+b) = \mathrm{Var}(X)$ for any real b and that for all $a \in \mathbb{R}$: $\mathrm{Var}(aX) = a^2 \mathrm{Var}(X)$. Moreover, if $\{X_i\}_{1 \leqslant i \leqslant n}$ are independent random variables, then

$$\mathrm{Var}(a_1 X_1 + \cdots + a_n X_n) = a_1^2 \mathrm{Var}(X_1) + \cdots + a_n^2 \mathrm{Var}(X_n).$$

If we have a sample of numbers $\{x_i\}_{1 \leqslant i \leqslant N}$ with arithmetic (sample) mean

$$\mu := \frac{1}{N} \times \sum_{1 \leqslant i \leqslant N} x_i,$$

then the *sample variance* is defined to be

$$\sigma^2 := \frac{1}{N} \times \sum_{1 \leqslant i \leqslant N} (x_i - \mu)^2 = \left(\frac{1}{N} \times \sum_{1 \leqslant i \leqslant N} x_i^2 \right) - \mu^2 = \frac{1}{2N^2} \times \sum_{1 \leqslant i,j \leqslant N} (x_i - x_j)^2.$$

The *standard deviation* of the collection of numbers is given by σ – the square root of the variance.

For two random variables X, Y we define their *covariance* to be $\mathrm{Cov}(X,Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$. The covariance is linear in each argument. We have that

$$\mathrm{Var}(\mathbb{1}_A) = \mathbb{P}(A)(1 - \mathbb{P}(A).$$

Similarly, the *correlation* between X and Y is defined to be $\mathrm{Corr}(X,Y) = \mathrm{Cov}(X,Y)/\sigma_X \sigma_Y$. For all real $a, c \neq 0$, $\mathrm{Corr}(aX + b, cY + d) = \mathrm{Corr}(X,Y)$. We also have

$$\mathrm{Cov}(\mathbb{1}_A, \mathbb{1}_B) = \mathbb{P}(A \cap B) - \mathbb{P}(A)\mathbb{P}(B).$$

17.3.3 Law of large numbers

17.4 Cauchy-Schwarz and L^p inequalities

We have that

$$\int_0^1 |f| \leqslant \left(\int_0^1 |f|^2 \right)^{\frac{1}{2}},$$

and more generally that

$$\left| \int fg \right|^2 \leqslant \int |f|^2 \times \int |g|^2.$$

If $\frac{1}{p} + \frac{1}{q} = 1 \iff p + q = pq$, then

$$\left| \int fg \right| \leqslant ||f||_p \times ||g||_q.$$

Minkowski's inequality provides that

$$||f + g||_p \leqslant ||f||_p + ||g||_p.$$

Also, for any measure μ and $\alpha > 0$, we have that

$$\mu\left(\{x : |f(x)| > \alpha\}\right) \leqslant \left(\frac{||f||_p}{\alpha} \right)^p.$$

17.5 Inclusion-exclusion and variants

17.6 Miscellaneous approximations and inequalities – See two sections of green notebook

17.6.1 Jensen's inequality

Recall that a function $f : \mathbb{R} \to \mathbb{R}$ is called *convex* if for all $t \in [0,1]$ and $x, y \in \mathbb{R}$, we have that

$$f(tx + (1-t)y) \leqslant tf(x) + (1-t)f(y).$$

If X is a random variable and f is convex, then

$$f(\mathbb{E}[X]) \leqslant \mathbb{E}[f(X)].$$

17.6.2 Average value inequalities for monotone functions

If $f, g : [a, b] \to \mathbb{R}$ are monotone functions of the same monotonicity, then

$$\frac{1}{b-a} \times \int_a^b f(x)g(x)dx \geqslant \frac{1}{(b-a)^2} \times \int_a^b f(x)dx \times \int_a^b g(x)dx.$$

The order of the inequality is reversed if f and g have opposite monotonicity on $[a, b]$.

17.7 Stochastic processes (and Markov chains)

TODO: See `https://en.wikipedia.org/wiki/Markov_chain#Formal_definition`

17.8 Martingales